BinProv: Binary Code Provenance Identification without Disassembly

Xu He
George Mason University
Fairfax, Virginia, USA
xhe6@gmu.edu

Shu Wang
George Mason University
Fairfax, Virginia, USA
swang47@gmu.edu

Yunlong Xing
George Mason University
Fairfax, Virginia, USA
yxing4@gmu.edu

Pengbin Feng
George Mason University
Fairfax, Fairfax, USA
pfeng4@gmu.edu

Haining Wang
Virginia Tech
Arlington, Virginia, USA
hnw@vt.edu

Qi Li
Tsinghua University
Beijing, China
qli01@tsinghua.edu.cn

Songqing Chen
George Mason University
Fairfax, Virginia, USA
sqchen@gmu.edu

Kun Sun
George Mason University
Fairfax, Virginia, USA
ksun3@gmu.edu

ABSTRACT

Provenance identification, which is essential for binary analysis, aims to uncover the specific compiler and configuration used for generating the executable. Traditionally, the existing solutions extract syntactic, structural, and semantic features from disassembled programs and employ machine learning techniques to identify the compilation provenance of binaries. However, their effectiveness heavily relies on disassembly tools (e.g., IDA Pro) and tedious feature engineering, since it is challenging to obtain accurate assembly code, particularly, from the stripped or obfuscated binaries. In addition, the features in machine learning approaches are manually selected based on the domain knowledge of one specific architecture, which cannot be applied to other architectures. In this paper, we develop an end-to-end provenance identification system BinProv, which leverages a BERT (Bidirectional Encoder Representations from Transformers) based embedding model to learn and represent the context semantics and syntax directly from the binary code. Therefore, BinProv avoids the disassembling step and manual feature selection in provenance identification. Moreover, BinProv can distinguish the compilers and the four optimization levels (`O0/O1/O2/O3`) by fine-tuning the classifier model with the embedding inputs for specific provenance identification tasks. Experimental results show that BinProv achieves 92.14%, 99.4%, and 99.8% accuracy at byte sequence, function, and binary levels, respectively. We further demonstrate that BinProv works well on obfuscated binary code, suggesting that BinProv is a viable approach to remarkably mitigate the disassembler dependence in future provenance identification tasks. Finally, our case studies show that BinProv can better identify compiler helper functions and improve the performance of binary code similarity detection.

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10.
https://doi.org/10.1145/3545948.3545956

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

binary program, provenance, compiler, optimization level, BERT

ACM Reference Format:
Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. 2022. BinProv: Binary Code Provenance Identification without Disassembly. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545948.3545956

1 INTRODUCTION

Cross-architecture and cross-compiler binaries of the analogous source code pose a challenge for the binary code similarity analysis [7, 10, 17, 31, 36, 41]. While the CPU architecture (e.g., x86 or ARM) determines the instruction set, compiler and optimization options change the composition of instructions. The induced instruction differences impede the effectiveness of similarity detection. Thus, it is critical to accurately uncover these underlying influential factors for the binary similarity analysis and other downstream tasks, such as malware detection, plagiarism detection, authorship identification, and vulnerability discovery [9, 14, 25, 32, 41]. The process of identifying the compilation environments is also referred to as provenance identification [32].

Existing provenance identification solutions are keen to extract features from assembly instructions and then input these features into machine learning (ML) based paradigms [14, 25, 32]. They usually collect three types of features, namely, syntactic features, structural features, and semantic features. The syntactic features count the appearance of a program's properties in assembly, such as instruction idiom, function signature, and the frequency of specific

opcodes. The structural features represent the control structure or data flow of a program, such as the control flow graph (CFG) and function call graph (FCG). The semantic features are extracted by more complex analysis, e.g., the graph-based combined features and ML-based embedding representations. With the above features, provenance identification can be framed as an ML based classification task.

However, these prior works have three major limitations: dependency upon inconvenient and inaccurate disassemblers, manual feature extraction, and coarse-grained identification. First, they rely on the existing disassembly and binary analysis tools (e.g., IDA Pro [12], Angr [38], and Ghidra [24]) to obtain assembly code and extract features, since binaries lack high-level abstractions and cannot provide any meaningful features, such as the function boundary. However, those existing tools cannot guarantee to generate accurate assembly or structure information, particularly, for stripped or obfuscated binaries [1]. Second, the features used in the existing models are often manually selected and processed based on the domain knowledge for a specific compilation and CPU architecture and hence may not work across different architectures. Thus, the feature extraction needs to be redone manually for a different architecture. Moreover, there is a trend to accommodate a large amount of features to enhance the semantic information but lack of interpretation as to which works. Consequently, it may cause prohibitive time and computational complexities [14, 32, 33]. Third, the existing approaches focus on distinguishing two sets of coarse-grained optimization information, namely, the low optimization level (including `O0/O1`) and the high optimization level (including `O2/O3`). However, they cannot accurately distinguish `O2` from `O3` or `O0` from `O1`. Note that `O2` and `O3` contain different sets of flags. For instance, `O3` has 15 more flags than `O2` in GCC 8.2.0. Distinguishing `O2/O3` is essential to precisely restore the compilation configuration.

In this paper, we propose BinProv, an end-to-end deep-learning-based framework that can identify compilation provenance by taking the binary code directly as the input into a BERT (Bidirectional Encoder Representations from Transformers) based embedding model [6, 19] and stacking a fully connected model to classify provenance. It is based on our observation that different compilation configurations would change the byte sequence significantly in the binary code, so that different byte sequence patterns may imply different compilation provenance. Since a different context of a byte in binary code represents different instructions in assembly code, we build a BERT-based embedding model that is capable of learning the contextual semantics in machine code. BinProv can totally avoid the dependence on disassemblers since it only uses binary code as input. Also, BinProv is architecture-agnostic and thus does not require domain knowledge of a target architecture to manually extract the features. Moreover, using the rich semantics in the embedding, BinProv can distinguish between four finer optimization levels (`O0/O1/O2/O3`).

We resolve three main challenges in identifying compilation provenance from binary code. First, without access to the assembly code, we cannot obtain the syntactic and structural features of assembly instructions or functions, which are critical for machine learning models to identify provenance. Instead, BinProv depends on retrieving the contextual semantics of the binary code, since the changes of the compilation provenance have impacts on the context of the binary code. It decomposes the `.text` section of a binary code into a series of fixed-length byte sequences (e.g., 512 bytes) to serve as the basic unit of provenance identification.

Second, the traditional classification models can hardly capture the key bytes and context from the sparse semantics of the byte sequences that only consist of `'0'` and `'1'` bits. To solve this challenge, BinProv builds a BERT-based embedding model that is capable of learning the dense contextual semantics of the byte sequence bidirectionally [6, 19]. Then, we stack a fully connected network to identify the provenance. Since the classic BERT model has a complex architecture that requires a huge amount of time and resource to learn a large number of parameters, we employ a transfer learning based training strategy to reduce the training cost of the BERT model in two steps. In the first step, we pre-train the embedding model using a Masked Language Modeling (MLM) task [19], which excels at training the model to learn the contextual semantics among the bytes in a byte sequence. In the second step, we fine-tune BinProv (including both the embedding model and the classification model) with specific provenance tasks, i.e., compiler identification and optimization level identification. Besides, BinProv can also be fine-tuned for other tasks, such as identifying the architecture and types of functions.

Third, the provenance of a single byte sequence may not correctly represent the provenance of a function or the entire binary code. First, a single binary may be linked from multiple object files, which could be compiled using different optimization levels. Second, the machine code of some functions may not change with optimization levels, such as the compiler helper functions (e.g., `frame_dummy` and `_start`) generated by the compiler. However, our measurement study on Github's real-world projects shows that 96% projects are prone to use the same optimization level to generate individual binary file (see Section 2.3). Based on the above measurement study, BinProv adopts the majority voting mechanism over the provenance results from multiple byte sequences to improve its identification performance on the function level and binary level.

We implement a prototype of BinProv [1] with PyTorch [26]. To evaluate BinProv's performance, we conduct extensive experiments on 7065 program binaries from the BINKIT dataset [17], which consists of 51 GNU software projects compiled using GCC and Clang over four optimization levels. We compare the accuracy of BinProv with two state-of-the-art approaches, and the experimental results show that BinProv can outperform all those approaches. For both compiler and optimization level identification, BinProv achieves 92.14%, 99.4%, and 99.8% accuracy at byte sequence, function, and binary level, respectively. Then, we analyze the influencing factors on the system performance, including the embedding model, binary length, sequence similarity, sequence location, and function type. Finally, we conduct two case studies on compiler helper function identification and binary code similarity detection.

In summary, our paper makes the following contributions.

- We develop a deep learning based framework called BinProv to identify the compilation provenance using the contextual semantics of binary code. It does not require disassembler or the manual feature extraction.

[1] https://github.com/Viewer-HX/BinProv

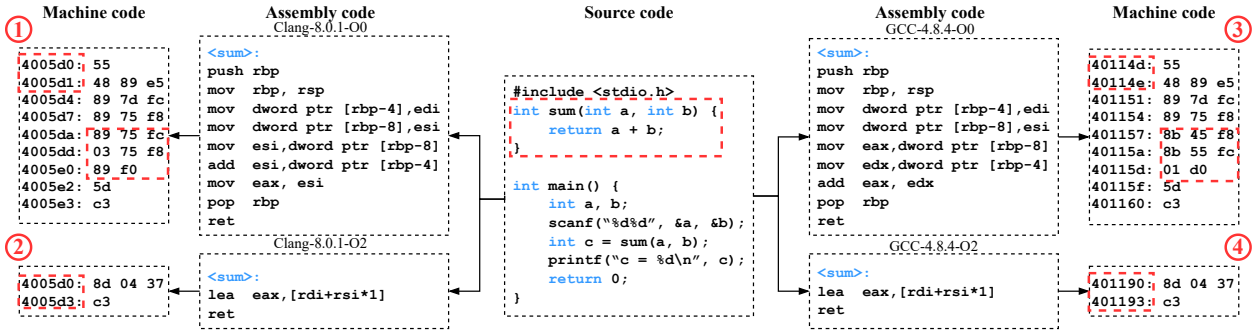


Figure 1: Four cases from source code to binary code under different compilers and optimization (opt) levels. Case 1 is compiled with `O0` under Clang 8.0.1; Case 2 is compiled with `O2` under Clang 8.0.1; Case 3 is compiled with `O0` under GCC 4.8.4; Case 4 is compiled with `O2` under GCC 4.8.4.

- We implement an architecture-agnostic prototype of BinProv, which can accurately identify different compilers and specific optimization level options.
- We evaluate BinProv on a massive binary dataset, showing that BinProv can achieve a high accuracy of provenance identification under various compilation configurations and improve the performance of binary code similarity analysis.

2 PRELIMINARY ANALYSIS AND MOTIVATION

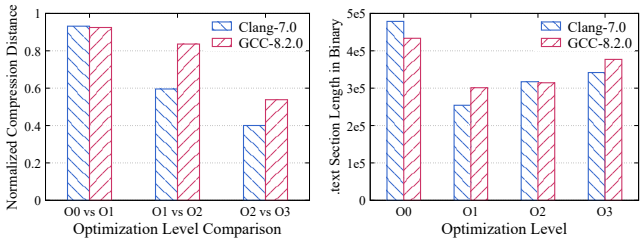
In this section, we first analyze the correlation between compilation provenance and binary code variance. We next investigate the limitations of existing disassemblers. Then, we study the usage of optimization levels in real-world projects and provide some key observations.

2.1 Code Variance from Compilation

The binary code variance is mainly caused by compilation in two aspects. First, compilers have their exclusive implementations that are determined by the instruction set architecture (ISA) [21], though they follow similar encoding rules. Second, different optimization options can cause even more changes. The compiler provides a number of optimization flags, which are grouped into a series of optimization level options.

Figure 1 shows four cases compiled using GCC 4.8.4 and Clang 8.0.1 with `O0` and `O2` options, respectively. We use them as examples to illustrate that the byte sequence intercepted from the machine code can embody the changes caused by compiler and optimization levels. Comparing cases① and ② (different optimization levels of Clang), the machine code under `O2` is much shorter than that under `O0`. It is the same for GCC between cases③ and ④. Comparing cases① and ③ (different compilers under `O0`), they have a similar construct in assembly code with 2 different registers and opcodes. For both provenance changes, the byte sequences in the machine code change drastically, as shown in the red dash squares.

Between cases② and ④ (different compilers under `O2`), they have the same assembly code and machine code; however, their storage address (offsets) are different. The start addresses are the same in cases① and② of Clang, while the start addresses are different in



(a) NCD between optimization levels. (b) Length vs optimization level.

Figure 2: The NCD distance and `.text` section length of binary variants under different compilers and optimization levels.

cases③ and ④ of GCC. This is because GCC relocates `sum(a,b)` behind `main()` when replacing `O0` with `O2`, while Clang does not. We see that the structural features cannot represent these address differences. Also, the statistical features cannot identify the operator changes in cases① and ③, since the opcodes are the same but in different orders. In contrast, the byte sequence can reflect such location changes in the machine code.

Specifically, the optimization level is provided to restructure the binary according to different optimization goals (e.g., compilation time, target file size, and execution efficiency) [11]. For GCC, `O0` does not perform any optimization; `O1` mainly optimizes code branches, constants, and expressions; `O2` focuses more on register-level and instruction-level optimizations; `O3` performs more optimizations based on `O2`, such as loop optimizations. The latter optimization level adds more optimization flags based on the former one, i.e., `O1`, `O2`, and `O3` add 43, 46, and 15 flags to `O0`, `O1`, and `O2`, separately. For the variances between all optimization levels, we find that differences between `O2`/`O3` are smaller than `O0`/`O1` and `O1`/`O2`, making it more difficult to distinguish `O2` from `O3`.

We also compare the binary length and similarity of the byte sequences from 235 binaries in the BINKIT dataset [17]. The similarity is measured by using normalized compression distance (NCD) [31]. As shown in Figure 2, the number and types of flags used in the optimization level affect the differences between byte sequences. The sequence difference between `O0` and `O1` is the largest (NCD>0.92),

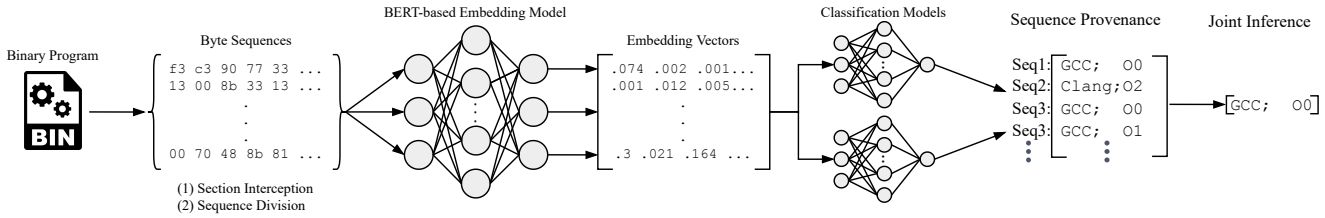


Figure 3: The architecture of BinProv.

and the length of binaries under O1 is the shortest. The major reason is that O1 involves 43 optimization flags, which reduce code size. The sequences are more similar between O2/O3 (NCD<0.58), and the binary length under O2/O3 becomes longer. That is because O3 only adds 15 flags but involves more complex constructs to increase the code size, such as overlapping blocks/instructions, inline data, and tail calls [1].

2.2 Error Propagation in Reverse Engineering

Researchers have used the structural and statistical features to represent the changes brought by compilers and optimization levels [14, 32, 33], but one prerequisite is to first obtain the assembly code via disassembly tools. However, the existing commercial and open-source disassemblers, such as Angr [38], IDA Pro [12], and Ghidra [24], have a long learning curve and cannot guarantee a satisfying accuracy, particularly, for the stripped and optimized binaries [1, 2, 5, 15].

A disassembler is mainly used for three tasks, namely, function boundary identification, assembly instruction transformation, and control flow graph extraction. Andriesse *et al.*[1] evaluate the performance of 9 disassemblers on x86/64 binaries and endorse that a higher optimization level can negatively impact the performance of disassemblers. Though IDA Pro offers the best performance on the above three tasks, it still cannot guarantee sufficient accuracy for each task. For x64 binaries on GCC at O1, IDA Pro can achieve up to 99% accuracy for all tasks; but at O3, the accuracy drops to 96% for assembly instruction conversion, 64% for function boundary identification, and 90% for control flow graph extraction. The trend is similar for the disassemblers on ARM architecture [15]. Recent studies [27, 35] even observe the worse performance of existing disassemblers in experiments, e.g., 58% accuracy for function boundary identification. Besides, obfuscation also challenges the effectiveness of disassembly tools [18]. The experiment of Jiang *et al.* [15] proves that the accuracy of the existing disassemblers reduces significantly on locating the instructions and function boundaries of the obfuscated binaries.

2.3 Optimization Levels in Real-world Projects

A large software project may be compiled with diverse configurations to generate binaries from different source code, including third-party libraries. For instance, we use GCC 8.2.0 to compile Linux kernel 5.8 and the build log shows that the entire kernel project is optimized with 2 O0 levels, 39,102 O2 levels, 50 O3 levels, and 453 Os levels. However, we find that the optimization level used to compile a single binary usually is the same, although different binaries in a project may choose different optimization levels. We compile the top 100 C/C++ projects on GitHub and Table 1 records the usage statistics of optimization levels in these projects. We see that most projects (90) use only one optimization level, though containing multiple source code files. In most cases, developers compile both their own code and upstream third party code with the same optimization options. Only 10 projects involve two or more optimization levels. Six of them generate multiple binaries, but each binary still uses one optimization option.

Table 1: Optimization level usage in top 100 C/C++ projects on GitHub.

# of Opt Levels	# of Projects	Distribution of Opt Options				
		O0	O1	O2	O3	Os
1	**90**	14	1	**62**	12	1
2	6	2	2	5	2	1
3	1	1	0	1	1	0
4	2	2	0	2	2	2
5	1	1	1	1	1	1

Binaries usually use one main optimization level (e.g., O2/O3) when using multiple options. O2 is the most common optimization level option, followed by O0 and O3. O1/Os are rarely used. Previous works focus on classifying low optimization level (O0/O1) and high optimization level (O2/O3), limiting their application in practice. To restore the compilation configurations in real-world projects, it is critical to distinguish the four optimization levels, particularly, between O2 and O3. From the perspective of optimization purpose, most projects focus on improving execution efficiency, but not the target file size and compilation efficiency. Therefore, they are prone to use O2/O3 rather than O0/O1. We conclude that most projects (>96%) are prone to use the same optimization level to compile individual binary file. Even when more than one optimization levels are used in an individual binary, one optimization level dominates the majority of the source code. Therefore, we can use majority voting to further improve the identification accuracy.

3 DESIGN OF BINPROV

The architecture of BinProv is shown in Figure 3. It consists of four stages: input pre-processing, embedding generation, classification, and joint inference. For pre-processing, BinProv intercepts and refactors the raw byte sequences from a given binary file. Then BinProv encodes the inputs as embedding vectors with a BERT-based embedding network, which aims to refine the semantics of byte sequences. Later, with the embedding representations, BinProv

utilizes a classification network to identify the provenance of byte sequences. In this embedding-classifying paradigm, we adopt a pretrain-finetune strategy. Based on the sequence-level provenance identification, we combine multiple sequences to identify binary-level and function-level provenance.

3.1 Input Pre-pocessing

As an end-to-end system, BinProv does not require complicated manual feature extraction and data pre-processing. It only needs simple operations to extract the formatted input sequences from raw binary programs, as shown in Figure 3. First, BinProv intercepts the `.text` section from the binary file so that BinProv can focus on the instruction-related byte sequences. This is because the `.text` section stores instructions and contains the most contextual semantics [40]. Then, we split the `.text` section into a series of byte sequences in hex (e.g. `0xcc`, `0x77`, `0xc3`, ...).

Byte sequence representation *b*. The input byte sequence is defined formally as a vector of hex values: $b = \{b_1, \cdots, b_i, \cdots, b_n\}$, where $b_i \in \{\texttt{0x00}, \cdots, \texttt{0xff}\}$. n is the sequence length, which is restricted by the processing capacity of the embedding model [6]. Meanwhile, we also balance the integrality of instructions. For ARM and MIPS architectures, each instruction is aligned by fixed length, e.g., 64 bit(8 byte). Moreover, since x86 instructions are variable-length. Thus, we take the longest sequence to increase the semantic richness and set the length n to a multiple of 8.

3.2 Embedding Generation

To better understand the contextual semantics, BinProv generates the embeddings for byte sequences. The contextual semantics is critical in the binary analysis since the same byte (or even byte snippet) in different sequences can be parsed into different operands or operators depending on their context. For example, byte `c3` can denote the opcode `ret` or the register `rbx` based on different context bytes [21].

In BinProv, we employ a BERT-based embedding model to learn the contextual semantics in byte sequences. Based on the transformer framework and multi-head self-attention mechanism, BERT can learn the bidirectional and long-range contextual semantics in the input sequences between each byte [39]. Besides, the training cost of BERT is affordable (around 80 million trainable parameters), compared with the state-of-the-art methods like GPT-3 (around 1 billion trainable parameters) [3]. Formally, the BERT model encodes the input byte sequence b into a composite embedding E, and updates E in the transformer f via back-propagation. The output embedding in each transformer is expressed as $E_{j+1} = f(E_j)$, where j denotes the j^{th} layer. The updated embedding is known as contextualized embedding [6, 22, 23, 28, 29]. The transformers in BERT share the same architecture, so the output of the BERT model can be expressed as $E_{final} = f_e(E(b))$, where f_e represents the whole embedding model, which consists of 12 transformers. $E_{final} \in \mathbb{R}^{N \times M}$, where N denotes the length of the input byte sequence and M denotes the dimension of the byte embedding.

Input representation *E*. In Figure 4, the input representation of the BERT-based model is a composite embedding E, composed of three parts: byte sequence E_b, segment sequence E_s, and position sequence E_p. The byte sequence $E_b = \{E_b(b_1), \cdots, E_b(b_n)\}$ is a hex vector converted from the binary input by the vocabulary dictionary. There is a total of 256 possible byte values in the vocabulary. The vocabulary also contains 5 extra tags (padding ⟨PAD⟩, start-of-sequence ⟨S⟩, end-of-sequence ⟨/S⟩, unknown ⟨UNK⟩, and mask ⟨MASK⟩) [26]. The segment sequence $E_s = \{E_s(b_1), \cdots, E_s(b_n)\}$ is defined to locate which binary program each byte belongs to, when a byte sequence contains multiple fragments from different programs. The position sequence $E_p = \{E_p(b_1), \cdots, E_p(b_n)\}$ denotes an integer sequence encoding the position of each byte in the binary program. The byte position is essential for understanding contextual semantics since swapping two bytes can significantly change the meanings of byte sequence.

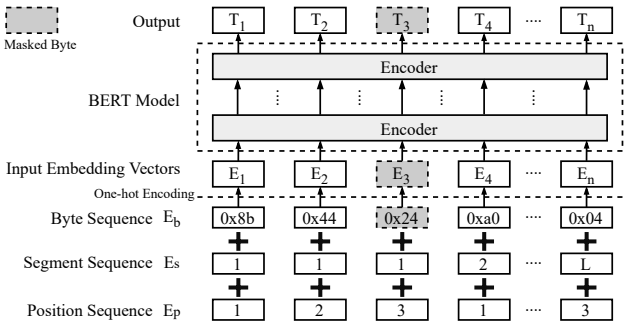


Figure 4: Pre-training of the BERT-based model on the masked language model (MLM) task. The grey dash square denotes the masked byte.

The segment and position embeddings can effectively assist BERT to understand the belonging and positional relations of each byte. As shown in Figure 4, we use one-hot encoding to generate byte embedding vectors for these 3 input sequences, and concatenate them into a unified input vector formulated as: $E(b) = f_{concat}(E_b(b), E_p(b), E_s(b))$.

Pre-training task. The BERT-based embedding model can be pre-trained with two tasks, i.e., masked language model (MLM) and next sentence prediction (NSP) [6]. MLM can help BERT learn bidirectional contextual semantics, while NSP boost the capability of BERT in sentence handling and paragraph matching. The latter is not the goal in this paper. In addition, Liu *et al.*[19] found that the BERT model pre-trained only with MLM can outperform that pre-trained with both tasks. Therefore, we adopt the MLM task to pre-train the BERT-based embedding model.

In Figure 4, the MLM task requires BERT to predict randomly masked bytes. In this task, BERT will learn the dependencies between the masked bytes and their context bytes hence learn the contextual semantics of the byte sequences. Assuming we mask t bytes randomly in a byte sequence b, the masked byte set is denoted as $m = \{m_1, m_2, \cdots, m_t\}$. We need to predict m using a pre-trained model $f_p(;\theta)$ in MLM, where f_p represents the BERT model stacking with a softmax layer to predict the masked bytes, and θ represents the trainable parameters in BERT. The objective of training f_p is to search for the optimal θ to minimize the cross-entropy loss between the inferred bytes ($\hat{m} = f_p(mask(b); \theta)$) and

the real bytes (m), which can be formally expressed as:

$$\arg\min_{\theta} \sum_{i=1}^{|t|} -m_i \log(\hat{m}_i). \quad (1)$$

By the gradient descent back-propagation, we can find the optimal parameters θ for Equation (1) so that BERT can learn the contextual semantics of byte sequences. Then, we remove the softmax layer and use the outputs of the last transformer E_{final} as the input features for provenance identification.

3.3 Provenance Classification of Byte Sequence

As shown in Figure 3, the third stage of BinProv is the classification. We decompose the provenance identification task into two sub-tasks: compiler type identification and optimization level identification. The task decomposition reduces the number of categories to be classified, specifically, 8 combination categories are split into 2 compiler categories (i.e., GCC, Clang) and 4 optimization level categories (i.e., `00/01/02/03`), respectively. We also find that specialized classifiers have better performance in the evaluation.

Both tasks share the same network architecture and only differ in their labels. With the semantically-rich embeddings, we use a two-layer fully connected neural network to identify the provenance of byte sequences. The first layer aims to reshape the input vectors and weaken the border weights of the embeddings. This layer is designed for the x86 architecture with variable-length instructions, which can lead to broken instructions in the sequence border. The second layer is equipped with a softmax function to predict the provenance for each sequence. Let f_c denote the classification model. Given an embedding E_{final}, the output is inferred by $\hat{y} = f_c(E_{final})$, where $\hat{y}$ can be specialized as $\hat{y}_c$ (compiler type) or $\hat{y}_o$ (optimization level). Overall, the whole forward mapping function $F(;\theta')$ in BinProv can be expressed as:

$$\begin{aligned} \hat{y} &= F(E(b);\theta') = f_c(E_{final}) \\ &= f_c(f_e(f_{concat}(E_b(b), E_p(b), E_s(b)))). \\ s.t. \quad & \hat{y} \in \{\hat{y}_c, \hat{y}_o\},\ \hat{y}_c, \hat{y}_o \in \mathbb{N} \\ & b \in \{\texttt{0x00}, ..., \texttt{0xff}\}^n \end{aligned} \quad (2)$$

where θ' represents the total parameters of the embedding model and the classification model.

Fine-tuning task. Based on the pre-trained parameters, we further perform end-to-end fine-tuning to precisely adjust parameters in order to fit with specialized downstream tasks. As depicted in Equation (2), the objective function of the fine-tuning task is to search for the optimal θ' that minimizes the cross-entropy loss. There are 3 differences between fine-tuning and pre-training. First, the parameter space is different ($\theta \neq \theta'$) due to the different structures of output layers. Second, the starting search points in their parameter space are different. Take the embedding model f_e as an example, the parameters in fine-tuning are updated based on the pre-trained values, while the parameters in pre-training are updated from random values. Finally, the size of training set in fine-tuning is much smaller than that in pre-training, because only the parameters of classification model (account for a small portion) need to be adjusted significantly during the fine-tuning phase.

Table 2: The statistics of binaries selected from BINKIT [17].

Dataset	# of Projects	# of Binaries	# of Arch.	# of Opt	# of Comp.
NORMAL	51	6280	4	4	2
OBFUSCATION	51	785	1	4	2

3.4 Joint Classification of Function and Binary

BinProv can identify the compiler type and optimization level of each byte sequence. However, any individual byte sequence can not represent a meaningful entity (e.g., function, binary program). On one hand, different parts of a binary may be compiled with different configurations, theoretically. On the other hand, the construct of partial functions may not change with provenance, such as the auxiliary function generated by the compiler. Hence, to infer the entity provenance, we need to aggregate the involved sequences for joint classification.

Our joint classification is based on the observation that sequences belonging to the same binary should have the same provenance with high probability. Therefore, it is valuable to adjust the overall inference by jointly using local consistency [32]. In this paper, we infer the provenance of a binary by majority voting over all sequences that belong to the same binary.

For the binary with diverse provenances, we utilize the majority voting result over the sequences belonging to the same function, because a single function can only have one type of provenance. Our evaluation shows that majority voting can significantly improve the performance of BinProv, compared with the direct classification over a single sequence.

4 IMPLEMENTATION

In this section, we first introduce the implementation environment and training settings of BinProv, and then discuss the dataset, baselines, and metrics used to evaluate BinProv.

4.1 BinProv Implementation

Environment. We build BinProv using PyTorch 1.8.1 with CUDA 11.2 and CUDNN 8.1.0. The neural network architecture of BinProv is conducted with the Fairseq toolkit [26]. The entire system is deployed on a Ubuntu 20.04 server with an Intel Xeon 5122 CPU and 512GB RAM. We train BinProv using 2 NVIDIA GTX 2080-Ti GPUs with 11 GB memory.

Training settings. Since we pre-train BinProv with the MLM task, we adjust the hyperparameter setting based on the existing schemes [8, 19] that only focus on MLM. For the masking strategy, we adopt the settings in Pei *et al.*'s work [27], because they also apply BERT on the binary analysis. We randomly select 20% bytes to mask in each sequence. In these randomly-chosen bytes, 50% are replaced with the ⟨`MASK`⟩ tags, and the other 50% are replaced with random hex values. Moreover, these masked bytes are different at each epoch, which means the random seed is reset for each epoch.

For the dataset division, we first construct the pre-training set by selecting at least one binary (2×4 variants) from each software project. Noted that the dataset we use includes 51 software projects (see 4.2). For the training and testing data splitting, we randomly

select different binaries to construct the fine-tuning set and testing set. The data-split ratio of fine-tuning and testing is 8:2. We further ensure no overlapping between training and testing sets so that our experiments can evaluate the real generalization ability of BinProv.

4.2 Experiment Settings

Dataset. In this paper, we use the BINKIT benchmark collected by Dongkwan *et al.*[17] to train and evaluate BinProv on the provenance task. The dataset contains binaries from 51 real-world GNU software packages. To our best knowledge, BINKIT has the largest capacity and the best diversity in all public benchmarks. BINKIT compiles the source code using 2 compilers (GCC and Clang) with 4 default optimization levels (`O0/O1/O2/O3`) under 4 architectures (x86_64, x86_32, ARM, and MIPS). In this paper, we focus on GCC and Clang because they are the most widely used cross-platform compilers and other compilers can only target few architectures. The statistics of binaries are shown in Table 2.

We mainly conduct the performance evaluation and comparison on binaries under the x86_64 architecture. This is because x86_64 is the worst case for BinProv due to the potential broken instructions near the sequence border. We also use BinProvto identify the architecture and test the performance of BinProv on other architectures. Besides the binaries compiled in normal mode, BINKIT also provides binaries compiled with other options, such as PIE (`-fpie`, `-pie`), no-inline (`-fno-inline`), LTO (`-flto`), and obfuscation (*Obfuscator-LLVM* [16]). The first three options are extra optimization flags, while obfuscation is a common anti-reverse engineering measure. If a binary is obfuscated, the disassemblers become more unreliable. Therefore, we also test the BinProv performance on obfuscated binaries.

In addition, we also compile some common algorithm programs [37] for the experiments in our case studies. The algorithm dataset includes a series of implementations of classical algorithms (341 programs), such as sorting and searching algorithms. Compared with BINKIT dataset, there are fewer functions and simpler call relations in binaries of the algorithm dataset, in which we can label easily the ground truth (e.g., the function type).

Baseline Solutions. We select two baseline methods from existing works, Origin [32] and O-glassesX [25]. Origin is the earliest ML-based provenance recovery approach proposed by Rosenblum *et al.* O-glassesX is the CNN-based provenance identification model proposed by Otsubo *et al.* in 2020. Both methods released the source code and claimed their identification accuracy exceeds 90% and publicly release research artifacts. Other recent method (e.g., [14]) does not provide executable source code. Because, neither baselines specifies the disassembly tool used, we utilize the *objdump* tool to disassemble binaries to extract after-disassembly features.

In addition, we construct two other methods for comparisons, i.e., BinRNN and O-glassesX*. For BinRNN, instead of using the embedding model, we convert the hexadecimal byte sequence into a binary sequence and deploy a RNN model for classification. Experimental comparison between BinProv and BinRNN can verify if the BERT model effectively learns the contextual semantics of the byte sequences. O-glassesX* is a variant of O-glassesX, requiring that the input bit sequences are converted from 16 instructions (16×128 bits). However, O-glassesX* skips the disassembly and directly intercepts the top 16×128 bits from the binary. O-glassesX* is used to verify if the performance of the baseline will degrade without disassembly.

Table 3: Accuracy comparison of BinProv with two baseline methods on the x86_64 architecture.

Basic task	BinProv w/o[2]	Origin [32]	O-glassesX [25]
Compiler (G/C)	95.47%	97.24%	**97.60%**
Opt level (H/L)	**98.90%**	96.14%	98.20%
Overall	94.77%	93.92%	**96.15%**

Metrics. We mainly use accuracy to measure the performance of BinProv and baselines, since the data distribution is balanced in our dataset. Accuracy is defined as the percentage of correctly classified samples over the total samples. We also use precision, recall, and F1 score to analyze the false predictions of different optimization level options.

Moreover, we use normalized compression distance (NCD) [31] to measure the degree of similarity between two byte sequences. NCD refers to the ratio of the saved space size by the maximum sequence size after compression. We use LZMA algorithm [13] to achieve lossless compress for the byte sequences in this paper.

$$NCD(x, y) = \frac{C(x \cdot y) - min(C(x), C(y))}{max(C(x), C(y))} \quad (3)$$

where $C(x)$ represents the size of sequence x after compression, while $x \cdot y$ indicates the concatenation of two sequences (x, y). A higher NCD means less similar between two sequences.

5 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of BinProv using massive real-world binary programs. We compare the accuracy of BinProv with those of baseline methods on different tasks and code granularities. Meanwhile, we evaluate the robustness of BinProv under different architectures and apply BinProv on extra compiler-related provenance and binary similarity detection tasks.

5.1 Accuracy of Provenance Identification

We first evaluate the performance of BinProv for the provenance identification task under the x86_64 architecture.

***a)* Binary classification of compiler and optimization level.** Table 3 shows the accuracy of BinProv and two baseline methods on two tasks, i.e., binary classification of the compiler (GCC/Clang) and optimization level (Low/High). Moreover, BinProv w/o means the model without majority voting; thus Table 3 depicts the results of individual sequences.

In Table 3, the accuracy of all methods on the compiler identification task exceeds 95%. The performance of BinProv w/o (95.47%) is not the best, but the gap with other methods is relatively small (around 2%). For the compiler identification task, the high accuracy mainly results from the apparent compiler patterns. For example,

[2]BinProv w/o means BinProv without majority voting.

Table 4: Performance of BinProv and two baseline methods on fine-grained classification tasks of optimization levels.

Opt level	BinProv w/o	Origin [32]	O-glassesX [25]
00/01/02/03	**91.07%**	-	-
00/01	**98.49%**	86.40%	89.20%
02/03	**83.64%**	51.60%	64%

"push ebp" always appears in the prologue of the functions compiled by GCC. Clang also leaves specific patterns in the compiled binary. For the optimization level classification task, all the approaches achieve higher accuracy (>96%). BinProv w/o (98.9%) outperforms other 2 methods, while Origin performs the worst (96.14%). The performance of optimization level identification also results from the significant differences between high/low levels. There are lots of distinct optimization flags between 00/01 and 02/03. For instance, 02 enables 52 more optimization flags than 01 in GCC 8.2.0 [11]. The binary constructs can be greatly changed by these added flags, such as `-falign-functions`, `-finline-functions`, and `-falign-jumps`. In Clang 6.0, 02 only changes 11 flags based on 01; however, the optimization target is changed. For example, the flags `-vectorize-loops` and `-vectorize-slp` generate vector instructions, which greatly increase the size of binary code.

***b)* Fine-grained classification of optimization levels.** Relying on the embedding model to extract contextual semantics, BinProv can uncover more subtle differences between optimization levels. Hence, we further extend the optimization level classification to 4 options i.e., 00/01/02/03. Table 4 shows the performance of BinProv and two baselines on 3 tasks with different granularities. Limited by their design frameworks, Origin and O-glassesX cannot make 4-class classification for optimization levels. However, we use these two baselines to distinguish 00/01 and 02/03, respectively.

In Table 4, the accuracy of Origin and O-glassesX drops significantly compared with their accuracy on the coarse-grained task (Table 3). For the 00/01 task, O-glassesX drops by 9% in accuracy while Origin drops by 10%. For the 02/03 task, the performance degrades more significantly. Origin reaches 51.6% (almost equivalent to random decision) and O-glassesX achieves 64%. In contrast, the performance of BinProv is much better. BinProv can achieve 91% accuracy on the fine-grained 00/01/02/03 task and 98.5% accuracy on the 00/01 task. Although the accuracy drops to 83.64% on the 02/03 task, BinProv still outperforms the other methods by 20% in accuracy. Our study find 74% of real-world projects mainly use 02/03 for optimization, thus it is vital to distinguish 02/03. As indicated by the results, distinguishing 02 and 03 is the most significant challenge in optimization level identification. 03 has 13 more optimization flags than 02 in GCC 8.2.0[11]; while 03 adds only 2 flags based on 02 in Clang 6.0 [4]. The difference between 02/03 is obviously smaller than that between 00/01 or 01/02.

Furthermore, we compare the precision, recall, and F1 score of BinProv for each optimization level in Table 5. For 00 and 01, BinProv performs well under both GCC and Clang. However, for 02 and 03, both the precision and recall are less than 85% due to the subtle differences between these 2 levels. Moreover, the 01 performance under Clang is worse than that under GCC since the precision is only 81% under Clang. This is because BinProv is prone to misclassify the sequences on 02/03 as 01 under Clang due to the high similarity. In addition, the precision of 02/03 under GCC is lower than that under Clang, because 02 and 03 share more optimization options in GCC. In general, the overall performance (F1 score) under GCC is better than that under Clang due to the following reasons. First, the number of training sequences under GCC is more than that under Clang, thus the embedding model can learn more semantics from GCC. Although we select the same number of sequences under GCC and Clang, binaries under GCC are generally longer than those under Clang by 12%. Second, Clang generates more auxiliary functions, which remain the same construct on different optimization levels. We will analyze the details of auxiliary functions in Section 5.2.

Table 5: Precision, recall, and F1 score of BinProv in optimization level identification task under GCC / Clang.

	GCC				Clang			
	00	01	02	03	00	01	02	03
Precision	99%	98%	74%	82%	96%	81%	85%	83%
Recall	99%	99%	77%	79%	98%	97%	72%	75%
F1 score	99%	98%	77%	79%	97%	88%	78%	79%

Table 6: Performance of BinProv and two baseline methods with majority voting at function and binary level.

	Provenance	BinProv w/[3]	Origin [32]	O-glassesX [25]
Func	Compiler (G/C)	**99.98%**	96.3%	97.60%
	Opt level (H/L)	**99.40%**	90.12%	98.20%
	Opt level (02/03)	**94.70%**	51.60%	64%
Bin	Compiler (G/C)	100%	100%	100%
	Opt level (H/L)	100%	100%	100%
	Opt level (02/03)	**99.8%**	60.1%	85.3%

***c)* Joint classification at binary and function level.** An individual sequence cannot represent the provenance of an entire function or binary, thus we aggregate the sequences to make joint inference. Table VI shows the results of BinProv, Origin, and O-glassesX at function and binary level aggregation. We compare their performance on 3 tasks: compiler classification, high/low optimization level classification, and 02/03 classification.

Comparing the results in Table 6 and Table 3, we find that the accuracy at the function and binary levels improves significantly via majority voting. For the function level, the performance of BinProv increases by 4.51%, 0.5%, 11.06% for these 3 tasks, respectively. On the contrary, the accuracy of Origin drops by 6.02% on the high/low optimization level task because Origin cannot make joint classification when only inputting a single function. For the binary level, these 3 methods can achieve 100% accuracy on the basic 2 tasks. Based on our observation c) in Section 2.3, our joint inference can apply to at least 96% of the programs. Moreover, the performance gain at the function level is lower than that at the binary

[3]BinProv w/ means BinProv with majority voting.

Table 7: The Robustness of BinProv under different settings (i.e., architectures and obfuscation).

Setting		Compiler(G/C)	Opt(02/03)	Opt(00/1/2/3)
ISA	x86_64	95.47%	83.64%	91.07%
	x86_32	96.22%	82.25%	89.24%
	ARM_64	98.80%	87.46%	93.76%
	MIPS_64	98.55%	87.91%	94.10%
OBF	SUB	93.21%	70.15%	75.60%
	BCF	95.03%	77.74%	83.28%
	FLA	94.66%	79.15%	84.33%

level. The number of sequences belonging to a function is less than those belonging to a binary; thus sequences belonging to a function may not contain enough semantics to imply structural changes. Also, some functions even do not change with different compilation configurations. In summary, joint prediction can further improve identification performance with a higher confidence..

***d)* Robustness across different architectures.** BinProv is an architecture-agnostic method that does not extract features based on the domain knowledge of specific architecture. We first fine-tuned our model to identify different architectures. we observe that BinProv achieve 99.99% accuracy on identifying x86, ARM, and MIPS architectures. Besides, O-glassesX [25] has similar performance.

Then, we compare the performance of BinProv on the compiler optimization level tasks with different architectures, as shown in Table 7. We find that BinProv achieves better performance under the ARM_64 and MIPS_64 architectures for all 3 tasks, because the fixed-length instructions can guarantee the integrality of sequences near boarder.

Besides, for the optimization level tasks, we find the performance under x86_32 is inferior to that under x86_64. We analyze binaries under these 2 architectures and have 2 observations. First, the number of instructions per function under x86_32 is nearly 90% more than that under x86_64. Second, there are more auxiliary functions under x86_32. For instance, the call `__x86.get_pc_thunk.bx` appears multiple times in each binary to redirect position-independent code. Therefore, the byte sequences under x86_32 may have multiple fixed components that do not change with the provenance.

***e)* Robustness to obfuscated binaries.** In Table 7, we also evaluate the BinProv performance on obfuscated binaries. The experiment dataset includes the binaries obfuscated by the *obfuscator-LLVM* tool [16]. We study three popular obfuscations techniques: instruction substitution (SUB), bogus control flow (BCF), and control flow flattening (FLA). SUB replaces the original opcodes with more complicated instruction sequences. BCF changes the function call graph by adding basic blocks. FLA disturbs the original CFG by creating fake control flows.

We find that the obfuscation has a limited effect on the compiler identification, where the performance fluctuation does not exceed 2%. However, in the optimization level tasks, the obfuscation causes significant performance degradation, especially for the instruction substitution (SUB). BCF and FLA techniques change the basic block relations to prevent reverse engineering from restoring the correct CFG. Meanwhile, BinProv does not rely on the CFG-based structural features, thus BCF and FLA are more robust to obfuscation (drop by 5%). Conversely, SUB only replaces original operators with functionally equivalent operators and hence breaks the contextual semantics. Hence, the performance with SUB obfuscation drops by around 15%, but BinProv still outperforms the two baselines.

Table 8: Performance of BinProv and two modified baselines using raw byte sequences.

Task	BinProv w/o	O-glassesX*	BinRNN
Compiler (G/C)	**95.47%**	91.4%	90.89%
Opt level (H/L)	**98.90%**	74.14%	63.20%
Opt level (02/03)	**83.64%**	54.1%	53.62%
Overall (Basic task)	**94.77%**	75.16%	64.64%

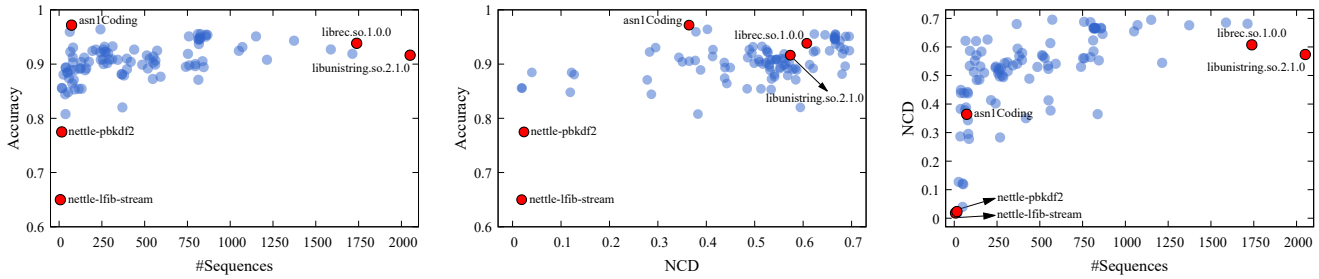
5.2 Impacts of Model Components

We also investigate the potential factors that may impact the performance of provenance identification, including the embedding model, binary length, sequence similarity, auxiliary function constructs, and sequence location.

***a)* Impact of the embedding model.** To evaluate the impact of the embedding model, we compare the performance of BinProv with two modified baselines, i.e., O-glassesX* and BinRNN. O-glassesX* omits disassembly and reshaping instructions, but inputs the raw byte sequence into the network of O-glassesX. BinRNN simplifies BinProv by replacing the BERT-based embedding model with a simple RNN structure.

Table 8 shows the performance of these modified methods on 3 tasks. Compared with O-glassesX and BinProv, both O-glassesX* and BinRNN have performance degradation on all tasks. The performance of O-glassesX* on compiler identification falls slightly by 6%, but the accuracy on optimization level tasks drops dramatically, especially for the 02/03 task (54.1%). This is because O-glassesX* surrenders the semantics from disassembly and only relies on the CNN-based encoders to extract semantics from the raw byte matrix. The CNN-based encoder model is inferior to the Bert-based embedding model in terms of learning semantics from the raw byte sequence. For the BinRNN without embedding model, the performance drops even more than O-glassesX*, which results from the semantic sparsity of raw binary sequence. The gap between BinProv and BinRNN indicates that the BERT-based embedding model in BinProv effectively eliminates the sparse semantic problems caused by the lack of disassembly.

***b)* Impact of binary length and sequence similarity.** Our preliminary analysis confirms that the optimization level has a significant impact on the length and similarity of the binary. We first measure the impact of binary length on the optimization level identification task. Figure 5(a) shows the accuracy on the binaries with different numbers of sequences. Each circle denotes a program with 4 binary variants (i.e., 00/01/02/03). We find that longer binaries tend to have higher accuracy in optimization level identification. Most of binaries with accuracy higher than 85% have more than 100 sequences. This means it has higher probability to correctly identify optimization levels if the binaries are



(a) #sequence vs accuracy. (b) NCD vs accuracy. (c) #sequence vs NCD

Figure 5: Impact of the binary length (#sequences) and NCD on the accuracy of optimization level identification task over the testing set (103 binaries). We annotate 5 programs as red circles, where 2 are the shortest binaries, 2 are the longest binaries, and 1 has the highest accuracy.

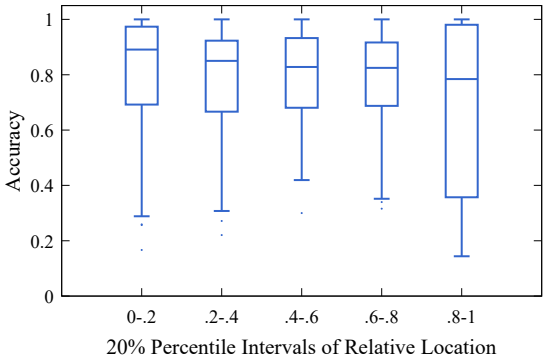


Figure 6: The accuracy of BinProv on sequences at different locations. The first bar represents the top 20% sequences in the binaries, and the other 4 bars denote later segments in the binaries.

long enough. We annotate 4 binaries with the longest and shortest length. The two shortest binaries coincidentally have the lowest accuracy (`nettle-lfib-stream`, `nettle-pbkdf2`), while the accuracy of the longest binary is relatively high (`librec.so.1.0.0`, `libunistring.so.2.1.0`).

We also assess the impact of sequence similarity (NCD) on the optimization level identification. Figure 5(b) depicts the relationships between identification accuracy and different NCDs. We can see that most of the binaries with high accuracy are clustered in the high NCD area. Therefore, it is easier to distinguish the optimization levels for the binaries with more distinct sequences. In Figure 5(c), there is a certain correlation between the length and similarity of the binaries (the Pearson correlation coefficient R = 0.611). The relative positions of the 4 red points are similar on Figures 5(a, b, c). Thus, longer binaries tend to have higher NCD while shorter binaries will have fewer byte changes. We argue that it is caused by the inherent auxiliary function constructs in binary.

***c)* Impact of the sequence location in binaries.** We analyze the performance for the sequences at different positions in a binary. We divide the binary sequences into 5 equal intervals based on relative position and evaluate the accuracy for sequences within each interval. Here, each binary in the testing set has at least 50 sequences.

As shown in Figure 6, the accuracy of optimization level identification gradually decreases from the head to tail of a binary. The median accuracy of sequences in the first interval (the top 20th percentile in position) is 89.1%. For the next 4 intervals, the median accuracy drops by 4.1%, 6.3%, 6.6%, and 10.7%, respectively. This is because the header sequences have more intact functions. We find 84.6% of the first sequences in our test set contain at least 8 intact functions. Therefore, header sequences usually contain relatively complete semantics. However, the deviation of the accuracy in the last interval (the last 20th percentile in position) is significantly greater than that of other intervals. This is because tail sequences tend to be shorter than other sequences and contain fewer intact functions.

***d)* Impact of functions with fixed construct.** There are three types of functions within the `.text` section. The first type is the user functions that are derived from the source code and responsible for the functional implementation. The second type is the library functions that are linked statically from the third-party library code. The third type is the helper functions that are derived from the compiler and responsible for starting up, initializing, and executing the entire procedure. For execution efficiency and security, most projects link library code dynamically rather than statically. Therefore, there are few library functions in the `.text` section. However, there are many compiler helper functions. The GCC generates 7 helper functions in the `.text` section, when compiling a C program that only contains an empty `main()` function.

Most helper functions have fixed functionalities and structures. To evaluate the impact of compiler helper functions, we first compare the function name and sequence similarity of 4,998 functions in 43 binaries compiled with GCC. We find that 247 functions (4.9%) have the same sequences on at least 2 optimization levels, which are partially listed in Table 9. For example, the structure of `_start` function does not change with the compiler and optimization level in all binaries. But its offset can be modified with the optimization level. Then, we also compare some helper functions with the same name but compiled by different compilers. GCC and Clang generate helper functions with the same function name but with different structure,

Table 9: Functions with fixed construct (partial samples).

Function name	Binary code	Assembly code	Compiler	Opt Level
_start	f30f1efa31ed4989d15e ···	endbr64; xor ebp, ebp; mov rdx, r9; pop rsi ···	GCC/Clang	O0/O1/O2/O3
register_tm_clone	be50404000488 1ee50404000488 9f0 ···	mov <value>, esi; mov <value>, esi; mov rsi, rax;	clang	O0/O1/O2/O3
register_tm_clone	488d3d792d0000488d35722d0000 ···	lea rip, rdi; lea rip, rsi; sub rdi, rsi;	GCC	O0/O1/O2/O3
frame_dummy	f30f1efae977ffffff0f1f8000000000	endbr64; jmpq <reg_tm_clones>; nopl 0x0(%rax)	GCC	O2/O3
frame_dummy	f30f1efae977ffffff	endbr64; jmpq <reg_tm_clones>	GCC	O0/O1
_dl_relocate_static_pie	f30f1efac3662e0f1f84000000000090	endbr64; retq; nopw cs:0x0(rax, rax, 1); nop	Clang	O0/O1/O2/O3
exp_search.part.0	440fb7c64889d1be01000000eb09 ···	movzwl si, r8d; mov rdx, rcx; mov 0x1, esi; ···	GCC	O2/O3

Table 10: The performance of BinProv and BinComp on the compiler helper function detection.

Method	Prov.	O0	O1	O2	O3	Mixed
BinProv	GCC	99.9%	99.9%	99.8%	99.9%	99.8%
	Clang	99.9%	99.9%	99.9%	99.8%	
BinComp [30]	GCC	99.8%	99.9%	98.6%	99.1%	73.7%
	Clang	99.7%	99.7%	99.6%	99.7%	

Table 11: The performance of the binary code similarity detection on the mixed and classified datasets.

Method	Top-1		Top-5	
	Mixed	Classified[4]	Mixed	Classified
TIKNIB [17]	74.5%	88.4%	81.3%	95.2%
Gemini [41]	64.2%	82.9%	72.2%	91.6%

such as register_tm_clones. Some helper functions are generated by specific compiler. For instance, _dl_relocate_static_pie function is generated only by Clang.

Some user functions with short sequence also remain the same structure when using the O2/O3 level, such as the function sum(a,b) in Figure 1. These user functions with fixed structure account for a higher proportion in small-size binaries. One existing approach [30] extracts helper functions by intersecting the disassemble functions across compiled binaries, which is inaccurate due to the user function with fixed structure. Therefore, we fine-tune BinProv to detect the compiler helper function and feed back the results into the identification of the compiler and optimization levels, which can significantly improve the previous performance.

5.3 Case Study I: Compiler Helper Function Detection

In this study, we fine-tune BinProv to detect the compiler helper functions from the user functions and library functions. We use the algorithm dataset [37] in this case study, because the ground truth (helper functions) can be verified more accurately when there are few user functions and simple library calls in programs. We use the binaries from the sorting programs (264 binaries consisting of 2,516 functions) to train the model and searching programs (88 binaries consisting of 709 functions) for the performance evaluation. We compare BinProv with the extraction approach used in BinComp [30], in which the compiler helper functions are obtained by intersecting the sets of disassembled binaries from different programs. The comparative results are shown in Table 10.

We first split binaries into 8 test sets in terms of the provenance. BinProv achieves very high accuracy (>99.8%) in all test sets. BinComp [30] also does well, since most compiler helper functions have fixed structures. However, BinComp can hardly detect some special helper functions. For instance, the function exponential_search.part.0 in program exponential_search is generated by GCC from a bigger function, in which some parts of the control flow could be inlined but GCC cannot inline the entire function. Therefore, it splits the function to leave the major part in its own function, which is named with the original function name plus .part.<number>, and inlines the rest in other functions.

We then compare BinProv and BinComp on the mixed test set. Given compiler and optimization levels unknown, the BinProv's performance is stable but the BinComp's performance falls dramatically (73.7%). BinComp randomly selects 2 binaries to intersect their assembly code, but the compiler conducts distinct assembly code even for helper functions under the same name. Besides, not all helper functions keep the same structure across different optimization levels. For example, the frame_dummy function has a different structure between O0/O1 and O2/O3. We attempt to select multiple programs and intersect them, whereas this method increases the accuracy as well as the time complexity.

After identifying the compiler helper functions, we apply the results to benefit the compiler and optimization level tasks. Most compiler helper functions are sensitive to the compiler family but stable to the optimization levels. In the compiler identification task, we only input the compiler help functions to BinProv. In the optimization-level recognition task, we remove all compiler helper functions and use the sequence of user functions and library functions as the input. Accordingly, the accuracy on the compiler identification task increases by 0.4%, and the accuracy on the optimization level task increases by 2.1%.

The identification of compiler helper functions can also help various tasks in binary analysis, such as authorship attribution, function recognition, and clone detection, in which filtering out the compiler help functions is a critical pre-processing step to reduce false positives [30].

5.4 Case Study II: Binary Similarity Detection

Existing work has confirmed that binary code similarity detection suffers from cross-platform binaries [7, 10, 17, 31, 36, 41]. In this

[4]Query and target functions are classified by BinProv.

study, we apply the identified provenance results into the binary similarity detection task. Before detecting similar binaries, we first use BinProv to determine the provenance of the query function. Specifically, we divide the target binary dataset into 8 target sets according to the provenance. We then detect similar functions in the target set consistent with the query function's provenance. Furthermore, we use BinProv to label compiler helper functions and remove them from the target binary dataset since they usually are not interesting target functions.

We employ two current binary code similarity detection methods, i.e., TIKNIB [17] and Gemini [41]. Both claim to effectively identify cross-platform binaries. TIKNIB uses a hill climbing approach to greedily select features for cross-platform binaries and calculate the similarity score based on the computation of the relative difference. Gemini uses the graph-neural-network-based method to generate embeddings for binary code similarity detection.

To ensure sufficient similar functions from similar source code, we additionally compile `coreutils 8.32` and `binutils 2.35`. Then we test the performance of TIKNIB and Gemini on the 1,690 binaries from `coreutils 8.24/8.32` and `binutils 2.29/2.35` (`coreutils 8.24` and `binutils 2.29` are included in the BINKIT dataset). There are 127,456 functions after removing the compiler helper functions. We select 500 different query functions, and the rest make up the target function set. Because the target task and model structure remain the same, we do not retrain the Gemini model. In the end, we compare their performance based on the average hit rate of queries under top-1 and top-5 values. In Table 11, we find that the performance of TIKNIB increases by 13.9% and 12.4%, for top-1 and top-5 hit rates, respectively. The performance of Gemini increases by 18.7% and 19.5%, respectively. Although the performance improvement of Gemini is higher than that of TIKNIB, Gemini still has lower hit rates than TIKNIB on both mixed and classified datasets. In general, BinProv significantly improves the performance of binary code similarity detection. The pre-classification using BinProv helps the similarity detection task to reduce the search space so that improve the matching possibility of target binaries.

6 DISCUSSION

The evaluation results demonstrate at least two advantages of BinProv. First, BinProv does not depend on any auxiliary information except plain byte sequences, thus it can eliminate the accumulated error from disassemblers and the complexity of feature engineering. In our experiments, the baselines are implemented based on the premise that the disassembly is accurate. As stated in Section 2.2, the disassembler accuracy drops as the optimization level increases. When the disassembler accuracy is 90%, the highest accumulative accuracy of baselines drops to 87.6% (O-glassesX), much lower than that of BinProv. Moreover, BinProv can still achieve a good performance on obfuscated binaries. Second, BinProv can classify optimization levels in finer granularity by subtly learning contextual semantics. Compared to previous works, BinProv extends the identification of the optimization level from high/low levels to four fine-grained options. In particular, `O2/O3` can be distinguished with a higher accuracy, which is the most commonly used in real-world projects. The accuracy of BinProv surpasses all baselines by around 30% on `O2/O3` identification.

BinProv has some limitations. First, the header and tail of sequences under x86 may be broken instructions, since we split the sequences based on the fixed length. Though we add a layer to weaken the weights of borders of the sequence, the performance of BinProv under x86 is still slightly worse than that under ARM and MIPS. Second, we use the `.text` section in binary as the input; however, non-code bytes may be mingled in the `.text` section, such as alignment bytes and padding bytes between functions [1, 32]. These non-code bytes have a negative impact on the representation of a binary's semantic; however, it is still a challenge to distinguish the data embedded in the code section. Third, attackers may adopt new countermeasures against BinProv. For instance, our experiments show that instruction substitution (SUB) is effective in hiding the original context semantics. Thus, when the SUB obfuscation can be applied multiple times, it becomes much harder to identify provenance [16, 17].

For future work, as we observe that developers often use extra optimization flags in addition to the optimization level in real projects (see Section 2.3), we plan to conduct a more detailed identification of optimization flags used during compilation, which may be useful for downstream binary analysis tasks. Moreover, BinProv could also be applied for other tasks. For instance, we can use BinProv to identify the third-party components from binary packages by checking the different provenances.

7 RELATED WORK

The compilation provenance identification task was first proposed by Rosesenblum *et al.* [33] in 2010, initially dedicated to identifying the source compiler of binary programs. Their approach is to extract statistical instruction segment patterns from binary code in each function [34]. Further, Rosesenblum *et al.* [32] designed Origin to identify more provenance (compiler and optimization level) based on ML-based models (e.g., SVM and CRF). They first captured the pattern of function entry points (i.e., idiom, a short sequence of instructions with wildcards). Their basic idea is that different compiler provenance usually means different idiom combinations. They also extracted the CFG as an auxiliary feature.

Rahimian *et al.* [30] proposed BinCamp, which builds a three-layer attribution model for the compiler provenance identification task using function-level features. They also extracted function fingerprinting features based on the compiler-related functions, and then built a neural network to identify the provenance. However, BinComp relies on compiler-related functions, which are useful for the compiler identification but against the optimization level identification. Otsubo *et al.* developed O-glassesX [25] to transform binary code into binary matrices and identify the provenance using a CNN-based image recognition approach. BinProv shares a similar design idea as O-glassesX, that is, pay little attention to the feature extraction process, but design a neural network structure with higher performance. However, O-glassesX still relies on a disassembly tool, while BinProv does not.

In industry, disassembly tools are also integrated with the provenance identification function, e.g., IDA Pro [12]. It mainly relies on

the signatures of binary code to retrieve the compilation information. However, the signature database currently relies on manual updates and they may fail when there is a slight signature difference between the query and target programs.

Recently, some methods attempt to construct the embedding representations for binaries via the graph neural networks (GNNs). For example, Massarelli *et al.* build an attributed control flow graph (ACFG) by representing each basic block as an embedding vector [20]. Yuede *et al.* propose a similar approach that embeds multiple features into the ACFG, including statistical features at the binary level and the function level [14]. Then they use the graph neural network (GNN) model to identify the provenance.

In the past decade, researchers have gradually extracted more complex features. However, in comparison with the earlier work (e.g., Origin), the performance improvement is not significant. Intuitively, we raise a hypothesize that the complex features from control flow and assembly instructions have limited correlation with provenance. However, the byte sequences may have the highest correlation with the provenance.

8 CONCLUSION

In this paper, we propose BinProv, an end-to-end compilation provenance identification framework. BinProv leverages a BERT-based embedding model to learn the contextual semantics in binary code. The only feature used in BinProv is byte sequences, which can be directly extracted from binary code. BinProv is capable of achieving very high accuracy for identifying the compiler and optimization level for binary code, strongly supporting our stance that the disassembler is not indispensable in provenance identification. We can therefore infer the optimization level more accurately. The accurate identification of high optimization levels (O2/O3) is valuable for discovering the complex binary constructs due to optimization, as it can provide guidelines for other binary analysis tasks in fine granularity. The provenance at the function or binary level is determined by the majority voting of the byte sequences from the same function or binary. BinProv achieves 96.85% and 99.8% accuracy at the function and binary levels, respectively, outperforming existing approaches. BinProv is trained with the pre-training and fine-tuning strategies based on the transfer learning. The pre-trained BERT-based embedding model provides a basic understanding of binary semantics. BinProv then can be fine-tuned for many other tasks, such as identifying the compiler helper function. Furthermore, BinProv benefits the binary code similarity detection by classifying the provenance of the query and target function.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their valuable comments. This work was supported in part by ARO Grant W56KGU-20-C-0008, NSF Grant CNS-2007153, the Commonwealth Cyber Initiative, and NSFC Grant 62132011.

REFERENCES

[1] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, USA, 583–600.

[2] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, USA, 845–860.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., USA, 1877–1901.

[4] Clang Team. 2020. clang - the Clang C, C++, and Objective-C compiler. https://clang.llvm.org/docs/CommandGuide/clang.html.

[5] Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, Manu Sridharan, et al. 2019. IDAPro for IoT Malware analysis?. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA, 15.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv e-prints* abs/1810.04805 (2018), 1–22.

[7] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium (NDSS'16)*. Internet Society, San Diego, CA, USA, 1 – 15.

[8] Facebook AI. 2020. RoBERTa implemented in Fairseq. https://github.com/pytorch/fairseq/blob/main/examples/roberta/README.md.

[9] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. Association for Computing Machinery, New York, NY, USA, 480–491.

[10] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, 896–899.

[11] GCC team. 2018. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[12] Hex Rays. 2008. IDA Pro. https://www.hex-rays.com/ida-pro/.

[13] Igor Pavlov. 2021. 7z format. https://www.7-zip.org/7z.html.

[14] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. Vestige: Identifying Binary Code Provenance for Vulnerability Detection. In *Applied Cryptography and Network Security (ACNS 2021)*. Springer International Publishing, Cham, 287–310.

[15] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 401–414.

[16] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection SPRO'15*, Brecht Wyseur (Ed.). IEEE, Firenze, Italy, 3–9.

[17] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* 1, 23 (2022), 1–23.

[18] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*. Association for Computing Machinery, New York, NY, USA, 290–299.

[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv e-prints* abs/1907.11692 (2019), 1–13.

[20] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*. Internet Society, San Diego, CA, USA, 1–11.

[21] MazeGen. 2017. X86 Opcode and Instruction Reference. http://ref.x86asm.net/coder64.html.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint* arXiv:1301.3781 (2013), 1–12.

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NIPS)*. Curran Associates, Inc., USA, 3111–3119.

[24] National Security Agency. 2019. Ghidra. https://ghidra-sre.org/.

[25] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. 2020. o-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment. In *Proceedings 2020 Workshop on Binary Analysis Research (BAR)*. Internet Society, USA, 1–12.

[26] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint* arXiv:1904.01038 (2019), 1–6.

[27] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*. Internet Society, USA, 1–18.

[28] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543.

[29] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint* arXiv:1802.05365 (2018), 1–15.

[30] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. 2015. BinComp: A stratified approach to compiler provenance Attribution. *Digital Investigation* 14 (2015), S146–S155. https://doi.org/10.1016/j.diin.2015.05.015 The Proceedings of the Fifteenth Annual DFRWS Conference.

[31] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 142–157.

[32] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*. Association for Computing Machinery, New York, NY, USA, 100–110.

[33] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE'10)*. Association for Computing Machinery, New York, NY, USA, 21–28.

[34] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code.. In *Proceedings of the 23rd national conference on Artificial intelligence (AAAI'08)*. AAAI Press, Chicago, IL, USA, 798–804.

[35] Sri Shaila, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. 2021. DisCo: Combining Disassemblers for Improved Performance. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*. Association for Computing Machinery, New York, NY, USA, 148–161.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, USA, 138–157.

[37] The Algorithms. 2021. Set of algorithms implemented in C. https://thealgorithms.github.io/c.

[38] UCSB. 2016. Angr. http://angr.io/.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS 2017)*. Curran Associates, Inc., USA, 6000–6010.

[40] Wikipedia contributors. 2021. Executable and Linkable Format — Wikipedia. https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=1047842416.

[41] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, New York, NY, USA, 363–376.